

**APPLICATION**  
**FOR**  
**UNITED STATES LETTERS PATENT**

**TITLE:           MECHANISM FOR LOSSLESS TRACING IN AN  
ARCHITECTURE HAVING A DELAY SLOT**

**APPLICANTS:   Adam H. LEVENTHAL and Bryan M. CANTRILL**

**32615**  
PATENT TRADEMARK OFFICE

"EXPRESS MAIL" Mailing Label Number: EL974017918US

Date of Deposit: November 14, 2003

# **MECHANISM FOR LOSSLESS TRACING IN AN ARCHITECTURE HAVING A DELAY SLOT**

## **Cross Reference to Related Applications**

[0001] This application is related to an application entitled “Mechanism For Lossless Function Entry And Return Tracing”, and an application entitled “Mechanism For Lossless Tracing In An Arbitrary Context”, both filed simultaneously herewith, the disclosure of which is hereby incorporated by reference in its entirety.

## **Background**

[0002] Analyzing the dynamic behavior and performance of a complex software system is difficult. Typically, analysis of a software system is achieved by gathering data at each system call and post-processing the data. Data is gathered at each system by placing a probe at locations of interest in the software (*i.e.*, instrumenting the software to obtain an instrumented program) and gathering data when the probe is encountered by the thread executing the instrumented program.

[0003] Probes are typically represented in the instrumented code as trap instructions. The location (*i.e.*, address) of each trap instruction is stored in a look-up table and associated with an original instruction (*i.e.*, the instruction that was replaced when the program is instrumented).

[0004] When a thread executing the instrumented program encounters a trap instruction, control is transferred to a trap handler, which calls into the tracing framework and performs the actions associated with the trap instruction. The trap handler then looks up the original instruction in the look-up table. The trap

instruction is then overwritten by the original instruction (*i.e.*, the original instruction is placed back in its original location within the code path replacing the trap instruction that was just executed). The tracing framework then single-steps the original instruction (*i.e.*, the original instruction is executed and then control is returned to the kernel). The original instruction in the code path is then overwritten by the trap instruction that was originally encountered by the thread. The thread then resumes executing the instrumented program.

[0005] In a system in which more than one thread is executing within a given instrumented program, a particular thread may not trigger a probe (*i.e.*, encounter a trap instruction) if the thread encounters the original instruction corresponding to a probe as opposed to the trap instruction. This situation typically occurs when a first thread encounters the trap instruction and overwrites it with a corresponding original instruction, and while this is occurring a second thread encounters the original instruction. In this scenario, the first thread calls into the tracing framework to perform the actions associated with the trap instruction, while the second thread executes the original instruction but does not call into the tracing framework. The aforementioned method for instrumenting a program is typically referred to as “lossfull” (*i.e.*, all the requested tracing information is not obtained because in certain scenarios, such as the one described above, a probe within a give code path may not be encountered by all executing threads)

[0006] Alternatively, the original instructions may be replaced with a reserved trap instruction, and when a thread executing the instrumented program encounters the reserved trap, all threads executing in the instrumented program are suspended while the thread that caused the trap single-steps the original instruction, which is temporarily written over the trap instruction, as defined above. Note that by suspending all the threads executing when the instrumented program when trap is encountered by one of the threads, the execution of the tracing framework is effectively serialized. After the thread has single-stepped

the original instruction, the reserved trap that was encountered by the thread is copied back over the original instruction in the code path. All threads in the executing in the instrumented program then resume executing the instrumented program. The aforementioned method for instrumenting a program is typically referred to as “lossless” (*i.e.*, all the requested tracing information is obtained because the threads executing the instrumented program encounter all the probes in the code path in which they are executing).

### Summary

[0007] In general, in one aspect, the invention relates to a method for tracing an instrumented program using a thread, comprising transferring control of the instrumented program to a trap handler to obtain an original instruction associated with a probe, loading the original instruction into a scratch space, setting a program counter to point to the scratch space, setting a next program counter to point to a next instruction, and executing the original instruction in the scratch space using the thread, wherein executing the original instruction results in placing the instrumented program in a state equivalent to natively executing the original instruction.

[0008] In general, in one aspect, the invention relates to a system for tracing an instrumented program, comprising a program counter configured to store a current address corresponding to a current instruction in the instrumented program, a next program counter configured to store a next address corresponding to a next instruction in the instrumented program, a scratch space arranged to store an original instruction, a thread configured to execute the instrumented program and the original instruction, and a trap handler configured to halt execution of the thread when a trap instruction is encountered, to obtain

the corresponding original instruction from a look-up table using an address of the trap instruction, and to set the program counter to the scratch space.

[0009] Other aspects of embodiments of the invention will be apparent from the following description and the appended claims.

### **Brief Description of Drawings**

[0010] Figure 1 shows a tracing framework architecture in accordance with an embodiment of the invention.

[0011] Figure 2 shows a look-up table layout in accordance with one embodiment of the invention.

[0012] Figure 3 shows a flow diagram in accordance with one embodiment of the invention.

[0013] Figure 4 shows a flowchart in accordance with one embodiment of the invention.

### **Detailed Description**

[0014] Specific embodiments of the invention will now be described in detail with reference to the accompanying figures. Like elements in the various figures are denoted by like reference numerals for consistency.

[0015] In the following detailed description of embodiments of the invention, numerous specific details are set forth in order to provide a more thorough understanding of the invention. However, it will be apparent to one of ordinary skill in the art that the invention may be practiced without these specific details. In other instances, well-known features have not been described in detail to avoid obscuring the invention.

[0016] The invention relates to method and apparatus for tracing an instrumented program. More specifically, the invention relates to a method and apparatus for lossless tracing of an instrumented program. In one or more embodiments of the invention, an architecture having a delay slot denotes an architecture having a program counter and a next program counter.

[0017] Figure 1 shows a flow diagram detailing the collection of data in accordance with one embodiment of the invention. Specifically, Figure 1 provides an overview of the process for collecting data for the buffer (116). Initially, source code (100) is written/obtained/generated that defines a tracing function (*i.e.*, a request to obtain certain data). More specifically, the tracing function defines which probes (112, 114) to enable within the instrumented program (115), and what actions that the tracing framework (106) is to perform when the probes (112, 114) are triggered (*i.e.*, when a thread executing the instrumented program (115) encounters the probe (112, 114)). In one or more embodiments of the invention, a tracing function may define one or more actions that the tracing framework (106) is to perform when a probe (112, 114) is encountered.

[0018] The source code (100) is typically associated with a consumer (101). Note that a consumer (101) may define one or more tracing functions. The consumer is a virtual client that sends requests, in the form of tracing functions, to the tracing framework (106) to obtain information about the instrumented program (115). Further, the consumer (101) also retrieves the requested information, which is stored by the tracing framework (106) in the associated buffers (116).

[0019] The source code (100) is subsequently forwarded, via the consumer (101) to a compiler (not shown), where the source code (100) is compiled to generate executable object code (102). The object code (102) is then communicated to a tracing framework (106). The tracing framework (106) includes functionality to execute the object code (102). Specifically, the tracing framework (106) interprets

the object code (102) and directs the probe providers (110) to activate certain probes (112, 114) within the instrumented program (115).

[0020] The probes (112, 114) gather the specified information from the instrumented program (115), as defined by the object code (102) derived from the actions defined within the source code (100), and forward the information (directly or indirectly) to a corresponding buffer (116).

[0021] In one or more embodiments of the invention, each probe (112, 114) in the instrumented program (115) is represented by a trap instruction. The address corresponding to location of the trap instruction within the instrumented program (115) is recorded in a look-up table along with the original instruction (*i.e.*, the particular instruction that the consumer would like to execute to obtain data). In one embodiment of the invention, the original instruction corresponds to an action that is to be performed when the probe (112, 114) is encountered. The action, as noted above, is typically defined by the consumer (101). In one embodiment of the invention, representing each probe as a trap instruction and generating a corresponding look-up table may be performed by the tracing framework.

[0022] Figure 2 shows a look-up table layout in accordance with one embodiment of the invention. As shown in Figure 2, the look-up table (200) includes one or more entries each of which may include an address field (201) storing the address of the trap instruction within the instrumented program (115) and an original instruction field (203) storing the original instruction. The look-up table (200) may also store additional ancillary information needed to specify the address.

[0023] Figure 3 shows a flow diagram in accordance with one embodiment of the invention. More specifically, Figure 3 shows a flow diagram detailing the mechanism for collecting data using a probe in accordance with one embodiment

of the invention. Each component in Figure 3 may be implemented by one or more software modules, hardware components, or any combination thereof. Further, each component shown in Figure 3 may be distributed across one or more processors.

[0024] In Figure 3, a program counter (301) stores a value corresponding to a current address in the instrumented program (115) in which the thread (300) is executing. When a probe (308, 310, 312), represented by a trap instruction, is encountered by the thread (300), the thread (300) transfers control to a trap handler (303). More specifically, when a probe (308, 310, 312) is encountered, a trap instruction is triggered which is subsequently handled by the trap handler (303). The trap handler (303) searches a look-up table (304), using the program counter (301) value, to obtain the original instruction associated with the probe (308, 310, 312). Those skilled in the art will appreciate that various machine architectures may require additional information, aside from the program counter, to generate an address which may then be used to obtain the original instruction. In this embodiment, the trap handler (303) includes functionality to obtain such information.

[0025] The scratch space (306) is typically a small address range or allocation of an address space, which is used to temporarily store the original instruction. In one embodiment of the invention, the scratch space (306) is located outside the kernel. Note that if the system upon which the tracing framework (106) is executing supports multiple threads in a process, then the scratch space (306) is allocated on a per-thread basis. In one embodiment of the invention, the trap instruction is no larger than the size of the smallest original instruction that must be replaced.

[0026] After obtaining the original instruction, the trap handler (303) sets the program counter (301) to point to the scratch space, and the thread (300)



subsequently executes the original instruction in the scratch space. After the thread (300) has completed executing the original instruction, the address in the next program counter (302) is copied into the program counter (301) and the next program counter is incremented by the size of one instruction. Note that because the next program counter (302) is set at the time the thread executes the trap instruction, the next program counter (302), typically, does not need to be set by the trap handler (303) prior to transferring control back to the thread (300). Note that if the original instruction is a control-transfer instruction, the next program counter (302) is copied into the program counter (301) and a target address (*i.e.*, the address of the instruction that is to be executed next as a result of the control-flow instruction) is loaded into the next program counter (302) prior to transferring control back to the thread (300).

[0027] Prior to the collection of tracing information, one or more probes (112, 114) are activated per a consumer (101) request. The activation of a probe, in accordance with one embodiment of the invention, also includes replacing the original instruction in the instrumented program (115) with a trap instruction, storing the address of the trap instruction and the associated original program in a look-up table. In one embodiment of the invention, the scratch space is allocated each time a thread is created.

[0028] Figure 4 shows a flowchart in accordance with one embodiment of the invention. More specifically, Figure 4 shows a flowchart detailing the steps that occur when a probe (*e.g.*, a trap instruction corresponding to a probe) is encountered by a thread executing the instrumented program. When a thread executing the instruction is encountered, the thread transfers control to an appropriate trap handler (Step 400). The trap handler calls into the tracing framework to perform tracing operations as required (Step 402). In one or more embodiments of the invention, the tracing framework performs a tracing operation and generates corresponding tracing information. The tracing

information may include, but is not limited to, an argument, a pointer value, a name of a system call, etc.

[0029] Continuing with the discussion of Figure 4, after the trap handler has made the appropriate calls into the tracing framework, the trap handler looks up the original instruction in the look-up table using the address of the trap instruction (and additional information as required) (Step 404). The original instruction is subsequently copied into a scratch space (which may be allocated on a per-thread basis) (Step 406). Those skilled in the art will appreciate that on architectures with split instruction and data caches (as opposed to an integrated cache), the instruction cache and the data cache must be kept synchronized so that the original instruction stored in the scratch space is consistent with the original instruction in the instruction cache. Thus, the trap handler (or a related process) must copy the original instruction into the scratch space in a manner that synchronizes the data cache and the instruction cache.

[0030] In one embodiment of the invention, if the invention is implemented on SPARC<sup>®</sup> Architecture, then the scratch space is initially set to a 64 byte-aligned memory address. SPARC<sup>®</sup> is a registered trademark of SPARC International, Inc. The original instruction is then obtained from the look-up table and copied into a floating-point register (*e.g.*, the first floating-point register (%f0)). A block copy instruction using the ASI\_BLK\_COMMIT\_S address space indicator is subsequently issued. The block copy instruction stores the contents of the floating-point register into the scratch space. Further, the block copy instruction invalidates copies of the data in all caches, if present. As a result, the block copy instruction maintains coherency between the instruction cache and the data cache.

[0031] In another embodiment of the invention, the entire instruction cache may be flushed prior to copying the original instruction in the scratch space. The

flushing of the entire instruction cache would force subsequent references to the instruction cache to go to an external cache that is always consistent with the data cache. In another embodiment of the invention, the instruction cache is flushed on a per-cache line basis. Specifically, the cache line(s) in the instruction cache in which the original address could be placed, would be flushed. The per-cache line flushing may be performed using a displacement flush.

[0032] Continuing with the discussion of Figure 4, once the original instruction has been copied into the scratch space, the trap handler sets the program counter to point to the scratch space (Step 408). The trap handler then returns control back to the thread that initially executed the trap instruction (Step 410). The thread then proceeds to execute the original instruction in the scratch space (Step 412). Those skilled in the art will appreciate that once the original instruction has been executed, the program counter is set to the next program counter. The thread then proceeds to continue executing the instrumented program. Executing the original instruction in the scratch space places the instrumented program in state that is equivalent to the state of the instrumented program, had the original instruction been natively executed.

[0033] In one or more embodiments of the invention, the original instruction is evaluated to determine if it is a control-flow instruction (*i.e.*, an instruction that affects the value of the program counter). A branch instruction, a function call, and explicit reading of the program counter itself are examples of the control-flow instructions. If the original instruction is a control-flow instruction, then instructions whose semantics depend on the location of the original instruction (*i.e.*, location dependent instructions, instructions that are affected by the value of the program counter), are emulated in the kernel. The program counter and the next program counter as well as any other state with the system that would be modified by the native execution of the traced location-dependent instructions are updated based on the results of the emulation.

**[0034]** The invention provides an efficient means for collecting information about an instrumented program. Specifically, the invention provides a means to collect tracing information in multi-thread environment without losing tracing information.

**[0035]** While the invention has been described with respect to a limited number of embodiments, those skilled in the art, having benefit of this disclosure, will appreciate that other embodiments can be devised which do not depart from the scope of the invention as disclosed herein. Accordingly, the scope of the invention should be limited only by the attached claims.